Localizing Vulnerabilities Statistically From One Exploit

Shiqi Shen
shiqi04@comp.nus.edu.sg
National University of Singapore

Aashish Kolluri
aashish7@comp.nus.edu.sg
National University of Singapore

Zhen Dong
zhen.dong@comp.nus.edu.sg
National University of Singapore

Prateek Saxena
prateeks@comp.nus.edu.sg
National University of Singapore

Abhik Roychoudhury
abhik@comp.nus.edu.sg
National University of Singapore

ABSTRACT

Automatic vulnerability diagnosis can help security analysts identify and, therefore, quickly patch disclosed vulnerabilities. The *vulnerability localization* problem is to automatically find a program point at which the "root cause" of the bug can be fixed. This paper employs a statistical localization approach to analyze a given exploit. Our main technical contribution is a novel procedure to systematically construct a test-suite which enables high-fidelity localization. We build our techniques in a tool called VulnLoc which automatically pinpoints vulnerability locations, given just one exploit, with high accuracy. VulnLoc does not make any assumptions about the availability of source code, test suites, or specialized knowledge of the type of vulnerability. It identifies *actionable* locations in its Top-5 outputs, where a correct patch can be applied, for about 88% of 43 CVEs arising in large real-world applications we study. These include 6 different classes of security flaws. Our results highlight the under-explored power of statistical analyses, when combined with suitable test-generation techniques.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → *Software and application security*.

KEYWORDS

Vulnerability Localization, Directed Fuzzing

ACM Reference Format:
Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. 2021. Localizing Vulnerabilities Statistically From One Exploit. In *2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21), June 7–11, 2021, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3433210.3437528

1 INTRODUCTION

Security vulnerabilities can remain unpatched for months after their initial disclosure [32, 34]. To reduce the window of exposure, it is useful to diagnose the "root cause" of the bug quickly. Automatic techniques which can help narrow down the vulnerability location, i.e. the program point at which a patch should be applied, have been a subject of a rich line of prior work [11, 14, 21, 25, 37, 41]. We call this the *vulnerability localization* problem. Localization techniques often work with just one given exploit demonstrating the vulnerability.

Prior work on vulnerability localization has used powerful binary analysis techniques, such as dependency or taint analysis and symbolic execution, on the given exploit program trace [27, 33]. In this work, we investigate a statistical approach. Specifically, our approach is based on statistical fault localization [38]. Statistical localization is a technique which was proposed almost two decades earlier in the context of general program debugging. However, its prime usage is in settings where extensive pre-existing test harnesses are available and often written by developers for functional or regression testing. This technique has received much lesser attention in the context of software vulnerabilities, where such test-suites are *not* available beforehand. Public vulnerability reports (e.g. CVEs) often contain just one exploit or crash input. Even in this setup, statistical localization does offer a unique and desirable feature at the outset: It only assumes access to a test-suite and makes almost no assumptions about the semantics of the target program. Once an appropriate test-suite is created, statistical localization can be used *without* any alternative sophisticated program analysis that may not scale to large programs.

Statistical localization works by assigning each program location a probability estimate or score, which measures how likely the program is going to be exploited if the instruction corresponding to the location is executed. A High score suggests that it is both sufficient and necessary for the instruction to be executed in order to reach the vulnerability and exploit it. Patching at the location is likely to eliminate the vulnerability while minimizing the impact on benign program behavior—we explain this reasoning more formally in Section 2.2. Such analysis is easy to implement as probabilistic scores can be calculated by profiling executed locations using off-the-shelf program instrumentation engines.

However, a naive application of statistical localization does not produce high-fidelity results on software vulnerabilities. The reason is somewhat fundamental—despite extensive work on statistical localization, one question has remained unresolved: *Under which input distributions should these probabilistic scores be estimated?* When working with one exploit, there is no test-suite to begin with, which is unlike the setting of regression or functional program testing. This issue is of central technical importance because the probabilistic quantities of interest can not be robustly estimated under arbitrary input distributions. We experimentally demonstrate

ASIA CCS '21, June 7–11, 2021, Hong Kong, Hong Kong

ACM ISBN 978-1-4503-8287-8/21/06…$15.00
https://doi.org/10.1145/3433210.3437528

that test inputs not designed explicitly with the objective of high-fidelity localization will often lead to *over-fitting*. The estimated scores will either be biased towards failing tests (exploits) or benign ones, depending on which of these dominate the provided test-suite. Ad-hoc or poorly chosen test-suites fail to distinguish between candidate locations, essentially assigning most of them the same score. The problem is acute in large applications in which more than a thousand locations may be observed in single program execution.

This paper presents a new solution to the problem of synthesizing a test-suite which accurately localizes vulnerabilities in large programs. The key insight is to avoid over-fitting and estimation bias, by synthesizing a specific kind of test-suite which we refer to as *concentrated*. A concentrated test-suite exercises a sufficient diversity and quantity of program paths in the neighborhood of the exploit path. The probabilistic quantities of interest can be estimated robustly using such a test-suite.

We propose a simple procedure to construct a concentrated test-suite from a single exploit. The procedure is a new form of directed fuzzing, which we call as *concentrated fuzzing* or ConcFuzz. Directed fuzzing techniques have witnessed rapid advances recently, however, their prime application has been for crash reproduction and patch testing [8]. Our work is the first, to the best of our knowledge, to propose its application for statistical localization. ConcFuzz is unlike other forms of directed fuzzing, which primarily aim to reach a particular location. The goal of ConcFuzz is to estimate the probability of each branch being executed in exploiting and benign runs. It generates inputs that exercise paths in the neighborhood of the path taken by the exploit to estimate these probabilities.

ConcFuzz makes minimal assumptions. It is simple to implement as it requires instrumentation of a small number of program points, without the need for any complex analysis. We implement our proposed techniques in a tool called VulnLoc, which can take stripped programs binaries and a single exploit for analysis. VulnLoc also contrasts prior statistical localization tools in this design aspect of minimizing assumptions. Prior localization techniques make many assumptions, such as the availability of program source code [22], a plethora of auxiliary code metrics beyond just source code [24], or extensive test-suites [19]. VulnLoc can be used for localizing any kind of bugs directly in binaries, making it usable anywhere in the software development and operational security pipeline.

Our main empirical finding is that VulnLoc scales to large applications and has high accuracy in localizing vulnerabilities. We evaluate VulnLoc on 43 CVEs on programs ranging from 10K - 2M LOC. We use a single exploit available from the CVE report and the program binary for each benchmark. We use developer patches as the ground truth for measuring the accuracy of VulnLoc. In about 88% of these benchmarks, VulnLoc pinpoints at least one location in its Top-5 ranked outputs, where a patch equivalent to the *developer patch* exists. Our results meet a high bar for user acceptability—in a prior user study with practitioners, over 90% of the respondents have indicated willingness to inspect 5 locations, and a Top-5 localization accuracy close to 90% satisfies expectations of 90% of the respondents [22]. Our techniques can be used to localize from crashes, exploits, or any other oracle of failure—our tested CVEs cover 6 common types of vulnerabilities, in many of which the correct location is far from the point of crash.

Contributions. We make the following contributions:

```
1    static int PixarLogDecode(TIFF* tif){
2      ...
3    +  /* Check that we will not fill more than what was allocated */
4    + if (sp->stream.avail_out > sp->tbuf_size){
5    +   TIFFErrorExt(tif->tif_clientdata, module, "sp->stream.avail_out > sp->
           tbuf_size");
6    +   return (0);}
7      // write into sp->stream
8        int state = inflate(&sp->stream, Z_PARTIAL_FLUSH);
9    }
10   // Function that cleans up pixarlog state
11   static int PixarLogCleanup(TIFF* tif){
12     ...
13     _TIFFfree(ptr);
14   }
```

Figure 1: A simplified code snippet of CVE-2016-5314 where the vulnerability location is not right before the crash point.

(1) We present VulnLoc, a vulnerability localization tool with minimal assumptions. It takes a vulnerable binary and an exploit as input and outputs Top-5 candidate locations.
(2) VulnLoc is the first work to propose directed fuzzing for localization. We propose a novel directed fuzzing technique called concentrated fuzzing (ConcFuzz), which explores benign and exploiting paths in sufficient diversity around the given exploit path. We demonstrate that the test-suites generated using ConcFuzz are able to avoid over-fitting to specific test inputs and allow distinguishing effectively between different candidate locations.
(3) We evaluate the efficacy of VulnLoc on 43 CVEs in large real-world applications. VulnLoc identifies the correct locations for about 88% of the CVEs within Top-5 locations in around 4 hours per CVE.

2 MOTIVATION & PROBLEM

We focus on the vulnerability localization problem in this work. It has been an important step in multiple applications [6, 23].

2.1 Problem

To keep the assumptions minimal, we only assume access to a vulnerable binary *Prog*, an exploit input i_e and the corresponding security specification. The specification is violated while executing *Prog* with i_e. Given the specification, we implement a *vulnerability oracle* to detect whether *Prog* gets exploited under a specific test case. A program crash can be an oracle for memory safety. For numerical errors (e.g., divide-by-zero), the processor provides hardware bits that can be checked. No source-level information is necessary to implement the oracle.

For the rest of the paper, we use the notion of *observed branch locations*. Concretely, when we execute the program binary with an input, we observe only the branch locations executed. Based on which branches are observed in the execution of specific inputs (generated via fuzzing), certain branches are predicted as *candidate vulnerability locations* by our technique. We deem a location as a *correct* or *actionable* location, if the basic blocks immediately preceding/succeeding the predicted branch can be modified to fix the bug. Developer-generated patches serve as the ground truth for correct fixes and we consider patches that are semantically

equivalent to the ground truth as correct. Note that our localization granularity is basic blocks, which is one of the most developer-preferred granularity levels highlighted by a prior user study [22].

For security bugs, one could assume that a location right before the crash point is sufficient to be a correct vulnerability location [16]. This is not a valid assumption, and in fact, developer patches often do not follow such a pattern (see our Section 6). The task of identifying a correct vulnerability location is more subtle. There are two objectives for patching at a correct location: 1) stopping the failing (or exploiting) program runs and 2) preserving compatibility with benign runs. Consider a bug in the open-source library LibTIFF [26] shown in Figure 1. It is a heap overflow vulnerability involving the buffer `sp->stream`. It arises from out-of-bound writes in `PixarLogDecode` function at Line 8 without checking the buffer length, which causes the head of the next heap to be filled with arbitrary data. The crash occurs when the invalid pointer is freed in another utility function called `_TIFFfree`. Since it is a utility function which is used by other functions (such as `PixarLogCleanup`), whether the pointer to be freed is valid or not at the crash location is *unknown*. Moreover, any changes to the utility function `_TIFFfree` would change other benign behavior of the program. Instead, a better way to fix the vulnerability is to prevent the out-of-bounds write itself since the buffer size is known during the write, as done in the developer patch. Extending beyond this example, such localization rules utilize domain-specific knowledge of the vulnerability type and variables in scope. We seek a procedure that does *not* rely on such domain- and context-specific analysis.

2.2 Background: Statistical Localization

Consider the execution of $Prog$ under i_e. Let the set of program branches encountered in the execution before violating the security specification be $V = \{v_1, v_2, \cdots, v_n\}$ and the execution trace be $U = \langle u_1, u_2, \cdots, u_m \rangle$. Each $u_i \in U$ is an instance of a branch $L(u_i)$ where $L(u_i) \in V$. Now, consider the execution of $Prog$ under a wide variety of inputs I, possibly generated by fuzzing with an exploit as the seed. Under each input, we see a subsequence of U in the execution trace with a subset of V observed. Each v_i and u_j are associated with a Bernoulli random variable X_i and Y_j respectively. Each random variable takes a value 1 if it is observed in that execution trace, else 0. Similarly, the violation of the security specification can be captured by a Bernoulli random variable C, which is 1 iff the program violates the security specification.

This abstraction allows us to reason about the statistical correlation between the events where C and X take on certain values. If an event X_i=1 happens in *all* of the exploit traces in the input set I, one could induce that (C=1) $\Rightarrow$ (X_i=1), which indicates that patching at v_i might avoid all exploits seen. However, it is possible that the patch might significantly change the benign behavior of the program. Conversely, consider an event such that whenever it occurs the program gets exploited. One can then induce that (X_i=1) $\Rightarrow$ (C=1). Since this event is not observed on any benign test, patching at v_i is likely to have the least impact on benign behavior, with the caveat though that such a patch may not cover *all* exploits seen. The best patch should prevent all exploits while having the least impact on benign runs. Overall, an event which is both necessary and sufficient carries a strong signal of the root cause underlying the exploit and is an ideal location candidate.

Consider the branch v_i such that the instances of v_i appear as $U_i = \{u_j | L(u_j) = v_i\}$. As a result, we can compute the probability of each branch location being witnessed as the probability that at least one of its instances is witnessed:

$$P(X_i = 1) = P(\bigcup_{u_j \in U_i} Y_j = 1)$$

We now compute two scores for each branch location v_i:

- Necessity score is $P(X_i = 1 | C = 1)$, the likelihood of observing at least one instance of the branch on an exploit;
- Sufficiency score is $P(C = 1 | X_i = 1)$, the likelihood of getting exploited on an input where at least one instance of the branch is observed.

Similar to the scores for each branch location, we can also define necessity and sufficiency scores of a single branch instance u_j which are $P(Y_j = 1 | C = 1)$ and $P(C = 1 | Y_j = 1)$ respectively. Branches with the highest K necessity and sufficiency scores are highlighted to the developer. The developer can then synthesize a fix in or around these locations.

2.3 Our Approach

The framework presented thus far is similar to the underpinning of a long line of works on statistical fault isolation [19, 38]. However, there is a central issue left unaddressed, which we study here: *under which input distribution should the probabilities be estimated?*

Consider computing the necessity score for each instance $P(Y_j = 1 | C = 1)$. As the probability of observing u_j deep down in the execution of an exploit may be very small, the probability estimates will *over-fit* the given test-suite if the test-suite only contains a few observations over u_j. The same phenomenon arises when computing sufficiency score, as most instances with very few (or no) observations in benign runs, leading to an artificially high score. In other words, an arbitrary test-suite is unlikely to *distinguish* between u_j appearing in the exploit trace for vulnerability localization.

Need for Sufficient Observations of $Y_j = 1/0$. The crux of our problem is to construct a test-suite which distinguish u_j with only an exploit input. To explain it, we factorize the necessity score into:

$$\begin{aligned} P(Y_j = 1 | C = 1) &= P_1 \times P_2 + P_3 \times P_4 \\ P_1 &= P(Y_j = 1 | C = 1, Y_{j-1} = 1) \\ P_2 &= P(Y_{j-1} = 1 | C = 1) \\ P_3 &= P(Y_j = 1 | C = 1, Y_{j-1} = 0) \\ P_4 &= P(Y_{j-1} = 0 | C = 1) \end{aligned} \tag{1}$$

The terms P_1 and P_3 differentiate the correlation of u_j to C and of u_{j-1} to C. A high ratio of P_1 to P_3 means that u_j and u_{j-1} are equally correlated to $C = 1$, as the exploit inputs always see the co-occurrence of u_j and u_{j-1}. However, a low ratio means that u_j is more likely to be observed when $C = 1$ than u_{j-1}, hence distinguishing them. It is thus important to construct a test-suite from which terms P_1 and P_3 can be estimated robustly. In particular, we need a test-suite with a sufficiently large number of exploit traces observing u_{j-1} and not observing u_{j-1} respectively.

This motivates the need for what we call a *concentrated* test-suite—a test-suite that has sufficiently many tests, both observing

Algorithm 1 Meta algorithm for concentrated fuzzing.

1: **Input:** Exploit input i_e, Execution trace U, Instrumented binary $Prog$
2: **Result:** Test-suite T
3: $T \leftarrow \emptyset$
4: **for each** $u_j \in U$ **do**
5: executeTillPrefix($i_e, u_j, Prog$)
6: **for** k from 1 to α **do**
7: i_m = mutate(i_e, u_j);
8: t_m = execute($i_m, Prog$);
9: $T \leftarrow T \cup \{t_m\}$
10: **end for**
11: **end for**

and not observing each u_j. The concentrated test-suite would explore paths in the neighborhood of the exploit trace. This is apparent in P_1 in Equation 1: It is the probability of observing u_j, conditioned on the fact that we have observed u_{j-1}. This can be seen as following the trace of the given exploit i_e up to u_{j-1} but not necessarily following the exploit trace after u_{j-1}.

To generate a concentrated test-suite, we propose a new form of directed fuzzing technique called *concentrated fuzzing*. Concentrated fuzzing is constructed in a principled way, and it carefully tries to avoid artificially biased test cases towards observing the events (i.e., $Y_j = 1$ and $C = 1$) which we will estimate from. This highlights the importance of having various kinds of test cases including the exploits, the benign cases, the cases reaching the crash location and the cases deviating from the crash location. From a test-suite created by concentrated fuzzing, we identify the correct locations for fixing the vulnerability and show the corresponding empirical results (see Section 6).

Remark. Our work is the first to utilize directed fuzzing as a solution for vulnerability localization. While heavy-weight alternatives such as those based on symbolic execution are possible solutions, these face challenges in scaling on binaries [18]. However, fuzzing is simpler to implement and scales. Other forms of fuzzing (e.g., AFL [39] and Honggfuzz [1]) optimize for different objectives from concentrated fuzzing. Their goal is to cover more program paths, whereas our goal is to explore only the neighborhood of a given exploit trace to find likely vulnerability location. We also compare our test-suite from concentrated fuzzing with directed fuzzing tools like AFLGo [8] in our empirical evaluation.

3 CONCENTRATED FUZZING

Constructing a concentrated test-suite is not straight-forward since the probability of reaching u_j which is deep inside the program is very low under most input distributions. To achieve this goal, we propose a new solution called *concentrated fuzzing* (ConcFuzz).

Algorithm 1 shows the high-level algorithmic sketch of ConcFuzz. It creates a set of inputs fully exploring each u_j (loop at Line 4) starting from u_1. In each iteration, the basic idea is to force the program execution to reach u_{j-1} (at Line 5), and then generate sufficiently many test cases that either reach u_j (stay on exploit path) or not (loop at Line 6). To ensure the reachability of u_{j-1}, many different strategies can be used—for instance, we could pick a sample from the set of test cases generated so far on which u_{j-1} was observed, and replay the execution. For simplicity, we run the program with the given exploit i_e to execute a *prefix* of the exploit trace upto u_{j-1}. Then, we mutate i_e to generate test cases, some of which will observe u_j. Notice that if these mutations are made arbitrarily, the execution may diverge off the prefix early, failing to observe u_{j-1}. Section 4 describes the details of a directed fuzzing approach where certain input bytes in i_e remain unchanged, such that the prefix up to u_{j-1} will be executed with high probability. Random mutations to other bytes are created to create sufficient many samples over the events $Y_j = 1$ and $Y_j = 0$.

```
1 void write(int size, char *writeArr){
2  for(int i=0; i<size; i++){
3    writeArr[i] = "A"; // <---- buffer overflow
4  }}
5
6 int write_array(int wsize, int msize){
7  char *writeArray;
8  if (wsize > 20)
9    return -1;
10 else {...}
11 if (msize <= 10)
12   writeArray = (char *)malloc(msize);
13 else
14   writeArray = (char *)malloc(2*msize);
15 write(wsize, writeArray);// write wsize characters into writeArray
16 ...}
17
18 int main(int argc, char **argv){
19 int a, c, tag;
20 ...
21 FILE *fp = fopen("input.txt", "r+"); // Read inputs from a file
22 fscanf(fp, "%d, %d, %d", &a, &c, &tag);
23 ...
24 if(tag == 1){ // Read from input file
25   read_array(c, fp);
26   ...
27   write_array(c, a);
28 } else if(tag == 2){ // Write array
29   write_array(c, a);
30 } else if(tag == 3){ // split the input file
31   ...
32   char array[10];
33   write(10, array);
34 }
35 ...
36 return 0;}
```

Figure 2: Code snippet for illustrating our fuzzing and ranking. There is a buffer overflow in function `write`.

An Illustrative Example. Figure 2 shows a hypothetical vulnerable program which has a buffer overflow in `write` function at Line 3. Let us assume that the exploit input available is i_e=(10, 15, 2) where the three bytes of i_e are inputs to the program and are read from an input file "input.txt" at Line 21. On execution of i_e, the function `write_array` is invoked with the arguments `wsize`=15 and `msize`=10. It is easy to check that an additional check `msize` $\geq$ `wsize` on the size of memory being allocated to the buffer `writeArray` at Line 11 is an ideal patch. Note that the vulnerability location is not close to the point of the buffer overflow (Line 3), where the variable `msize` is not even in scope. Ideal vulnerability location can be far off from the exploit point, as in this example.

To pinpoint the correct location, ConcFuzz generates a concentrated test-suite to explore each on-exploit branch instance u_j. Recall that the goal of vulnerability localization is to identify the right location v_i for fixing the vulnerability rather than the instance u_j. Thus, in Figure 3, we summarize the number of test cases generated for each v_i in the concentrated test-suite. Notice that there

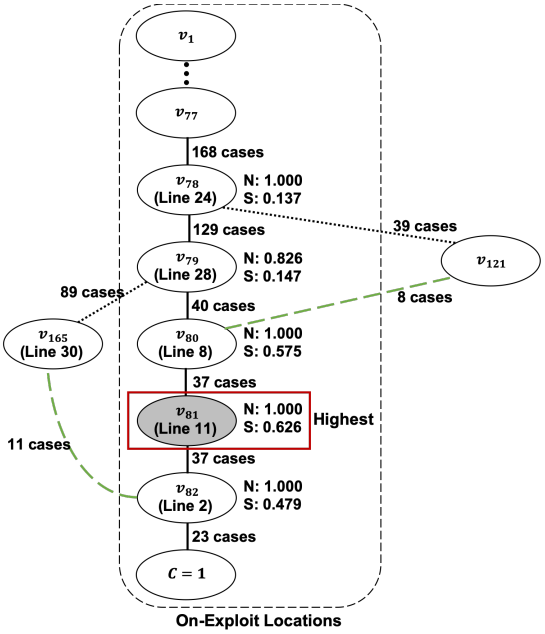

Figure 3: An example of concentrated test suite generated for the the code snippet in Figure 2 and its corresponding necessity (N) and sufficiency (S) score.

are many test cases for both observing and not observing each on-exploit location. These tests are not generated with the objective of causing an exploit. Biasing towards following $v_i, v_{i+1}, \cdots, v_n$ would skew the samples. Over-fitting to exploits and its effects are shown experimentally in Section 6.3. This is an important difference to recent works which explicitly aim to follow the path *suffix* of the exploit after v_i [18, 31]. Our tests seek to follow the given exploit path *prefix* up to v_i and then diverge.

Under the exploit i_e, the vulnerable program executes the branch at line 24 (v_{78}), 28 (v_{79}), 8 (v_{80}), 11 (v_{81}) and 2 (v_{82}) which are on-exploit locations. Taking v_{80} as an example, ConcFuzz forces the execution of the program to observe branches $\{v_1, \cdots, v_{79}\}$ and gets 48 cases observing v_{80} and 89 cases that do not observe V_{80} (but instead observes v_{165}). Among these 48, 23 cases trigger the buffer overflow while the remaining 25 cases do not. ConcFuzz generates sufficient test cases for each v_i in this way and finally computes the necessity and sufficiency score. As shown in Figure 3, v_{81} has the highest scores compared with the other on-exploit locations, which is the correct location for fixing the vulnerability.

Careful readers may notice that our procedure executes i_e to force the prefix up to u_{j-1}, when generating the concentrated test cases for u_j. This corresponds to the conditional probability $P(Y_j = 1|C = 1, Y_1 = 1, \cdots, Y_{j-1} = 1)$, as opposed to $P(Y_j = 1|C = 1, Y_{j-1} = 1)$ desired in Section 2. Note that the latter is a marginal

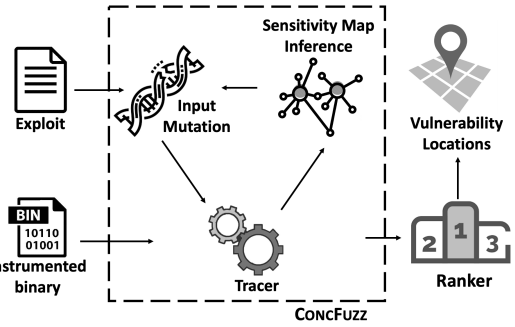

Figure 4: VulnLoc's Architecture.

probability which is a summation over exponentially many conditional probabilities:

$$\sum_{q_1 \in \{0,1\}} \cdots \sum_{q_{j-2} \in \{0,1\}}$$
$$P(Y_j = 1|T_1 = q_1, \cdots, T_{j-2} = q_{j-2}, C = 1, T_{j-1} = 1)$$
$$\times P(T_1 = q_1, \cdots, T_{j-2} = q_{j-2}|C = 1, T_{j-1} = 1)$$

Since measuring the marginal would require sampling across all paths leading up to u_j, estimating it may require intractably many test cases. However, as our experiments demonstrate, the $P(Y_j = 1|C = 1, Y_1 = 1, \cdots, Y_{j-1} = 1)$ serves as a good proxy for the marginal we desire. To understand why such a proxy works well in practice, let us consider the situation where there is conditional independence: the probability of observing an event, conditioned on having reached the point which observes u_{j-1}, is independent of the probability of reaching u_{j-1} in the first place. In such a case, $P(Y_j = 1|C = 1, Y_1 = 1, \cdots, Y_{j-1} = 1)$=$P(Y_j = 1|C = 1, Y_{j-1} = 1)$. In our example, the probability of reaching Line 11 is indeed almost independent of the probability of causing exploits—for any given value of `msize`, there are many values of `wsize` that would lead to exploits. Intuitively, when the vulnerability is dependent on a small number of branches, such "locality" creates conditional independence of the form that our technique works well with.

4 TECHNICAL DETAILS

The overall architecture of our tool called VulnLoc is shown in Figure 4. VulnLoc includes two main components: the concentrated fuzzer ConcFuzz and the ranker. ConcFuzz takes in an instrumented vulnerable program and an exploit input. It runs in a cycle until a pre-defined timeout is reached or sufficient test cases for each u_j have been generated. Given the concentrated test-suite, the ranker then computes the sufficiency and necessity score and reports the Top-K location with the highest combined score.

4.1 ConcFuzz Internals

Concentrated fuzzing, the high-level design of which is outlined in Algorithm 1, generates a concentrated test-suite by executing the prefix of each u_j. The key challenge is to generate multiple inputs which follow the prefix of u_j, since a random mutation of the exploit input i_e is unlikely to execute a long prefix. To tackle this problem, the main observation we make is that usually a small

Algorithm 2 The implementation of concentrated fuzzing.

1: **Input:** Exploit input i_e, Instrumented Binary $Prog$
2: **Result:** Test-suite T
3: Seed Pool $SP \leftarrow \{i_e\}$; $T \leftarrow \emptyset$
4: **repeat**
5: $i_s \leftarrow$chooseSeed(SP)
6: $t_s \leftarrow$execute($i_s, Prog$)
7: Sensitivity Map $SM \leftarrow initSM(i_s, t_s)$
8: Number of Mutated Bytes $\#B \leftarrow 0$
9: **repeat**
10: $\#B \leftarrow \#B + 1$
11: **repeat**
12: Bytes $B \leftarrow$selectMutateBytes($SM, t_s, \#B$)
13: **if** B is empty **then**
14: **Break**;
15: **end if**
16: **for** j from 1 to γ **do**
17: $i_m \leftarrow$mutate(i_s, B)
18: $t_m \leftarrow$trace($i_m, prog$)
19: $T \leftarrow T + \{t_m\}$
20: **if** i_m is an exploit with a new trace **then**
21: $SP \leftarrow SP + \{i_m\}$
22: **end if**
23: $SM \leftarrow$updateSM(SM, t_s, I_k, t_m)
24: **end for**
25: **until** Timeout reaches or all the branch instances have sufficient test cases
26: **if** $\#B \geq \beta$ **then**
27: **Break**;
28: **end if**
29: **until** Timeout reaches or all the branch instances have sufficient test cases
30: **until** Timeout reaches or all the seeds are checked

set of input bytes are responsible for observing each u_j. Thus, if we can fix the values of the bytes influencing all of $\{u_1, \cdots, u_{j-1}\}$ to their corresponding values in the exploit trace, then any mutation made on the remaining bytes will result in a new test case that most likely executes the prefix of u_j.

Precisely, let each input $i \in I$ contain q bytes of which each is represented as b_k ($k \in \{1, 2, \cdots, q\}$). ConcFuzz keeps track of the *influence* of different inputs bytes on each u_j. If we can observe that a change in the value of an input byte b_k results in a change on the state of u_j, we say that b_k influences u_j, or u_j is sensitive to b_k. The sensitivity relation is thus a binary relation $SM : (U, I) \rightarrow \{0, 1\}$. $SM(u_j, b_k) = 1$ if we have observed in concrete runs that u_j is sensitive to input byte b_k, and 0 otherwise. The relation is stored explicitly in a data structure called "sensitivity map". Note that the sensitivity map captures both control and data dependencies between branches and the input. This notion of influence or sensitivity is as same as that described in recent fuzzing works [10, 13], however, the way it is used in concentrated fuzzing is quite different. Algorithm 2 explains how ConcFuzz generates a concentrated test-suite with the help of the sensitivity map.

Sensitivity Map Inference. The sensitivity map is constructed directly from the observations during the execution of test inputs. We mutate each byte b_k of the input a constant number of times (see γ at Line 16) and observe whether the state of u_j changes due to each mutation. If we cannot observe u_j on the trace after the

```
1 // arr holds the fuzzer input
2 int buggy(char *arr){
3   b = arr[0]; c = arr[1]; d = arr[2]; e = arr[3]; g = arr[4]; ans = 0;
4   ...
5   if (b==1 && c==4){
6       ...
7   }
8   if (d==2 || e==3){
9       ...
10      if (g<5){
11        ... //Bug here
12      }
13    }
14  ...
15  return ans;}
```

Figure 5: The && condition on Line 5 requires single-byte mutation to infer that both $arr[0]$ and $arr[1]$ influence the condition. However, the condition on Line 8 requires the mutation on both bytes $arr[2]$ and $arr[3]$ simultaneously to infer that those bytes influence that condition.

mutation over an input byte b_k in i_e, we infer that b_k implicitly or explicitly influences the state of u_j i.e., $SM(u_j, b_k) = 1$ as well.

In more detail, for each round of fuzzing, ConcFuzz first compares the trace t_s under a selected exploit input i_s with the trace t_m under each mutated input i_m. If t_m diverges from t_s at u_j, ConcFuzz marks u_j as being sensitive to the input byte in i_s mutated to create i_m in the sensitivity map. To exemplify, let us revisit the running example from Figure 2. Given the selected exploit input $i_s = (10, 15, 2)$ and the mutated input $i_m = (10, 25, 2)$—where the first byte b_1 has value 10, the second byte b_2 is mutated and the third byte b_3 has value 2—ConcFuzz infers that the branch at Line 11 is sensitive to b_2 of the input. This is because branch at line 11 is not observed when $b_2 > 20$ in the trace of the exploit input i_s, and gets observed in the trace of the mutated input i_m.

ConcFuzz incrementally computes a sensitivity map over each mutation for each u_j. It starts with an empty sensitivity map (at Line 7) where each u_j is not influenced by any input byte. Then, ConcFuzz keeps updating it given a new test case generated for each round of fuzzing (at Line 23) until the timeout reaches or all the branch instances are fully explored.

Note that there could be scenarios where multiple bytes of the input may influence the same branch and the sensitivity map inferred by single byte mutations, as described above, may not be useful for generating concentrated test-suite. For instance, consider the code in Figure 5. The exploit input constitutes of $arr = [1, 4, 2, 3, 0]$ which satisfies the conditionals on Lines 5, 8 and 10. The first conditional statement (on Line 5) is a conjunction of conditionals on the values of two input bytes $arr[0]$ and $arr[1]$. Here, mutating one input byte at a time is sufficient to infer that the branch is sensitive to both bytes. However, this is not the case for the disjunction on Line 8. In this case, single byte mutation will lead to a conclusion that bytes $arr[2]$ and $arr[3]$ do not influence the branch as one of them is always $True$ while the other is being mutated. Therefore, while exploring the branch on Line 10, our sensitivity map will show that only $arr[0], arr[1]$ as important to execute the prefix till Line 8. Hence, VulnLoc would fix only the values of $arr[0], arr[1]$ and mutate other bytes including $arr[2], arr[3], arr[4]$. Consequently, the generated concentrated test-suite will have very few traces that

reach Line 10 attributing to a small random chance that the traces have passed condition on the prefix branch (Line 8). Therefore, we need to mutate both the corresponding bytes ($arr[2], arr[3]$) simultaneously to infer that the branch is sensitive to those bytes.

Remark: Approximation vs. Completeness. In general, sensitivity map inference is intractable since the number of combinations of such multi-wise mutations can be exponential in the size of input. Our sampling-based technique does not aim to capture all possible dependencies. To remain tractable, ConcFuzz performs multi-wise mutations iteratively (at Line 12 in Algorithm 2). It mutates one byte at a time in the first iteration and two bytes at a time in the second, until a user-configured number of iterations (default value =2). Our inferred sensitivity map, though necessarily approximated, is sufficient for achieving the accuracy reported in our evaluation.

Concentrated Test-suite Generation. The sensitivity map helps to generate sufficiently many samples for both events $Y_j = 1$ and $Y_j = 0$, having observed u_{j-1}. ConcFuzz first extracts all the input bytes from the sensitivity map to which $\{u_1, \cdots, u_{j-1}\}$ are sensitive. It forces these input bytes to take the same value as the selected exploit input i_s. Then, to obtain enough samples for $Y_j = 1$, ConcFuzz mutates the input bytes to which u_j is non-sensitive in i_s as the mutation over the non-sensitive bytes likely does not change the state of u_j. In contrast, for generating enough samples for $Y_j = 0$, ConcFuzz mutates bytes to which u_j is sensitive and keeps the non-sensitive bytes the same. For each round of fuzzing, ConcFuzz selects an instance u_j which is the earliest observed on the exploit trace but does not have sufficient test cases. Then, it follows the above approach to generate sufficiently many test cases with a limited number of mutations (from Line 9 to 29).

For example, let us revisit the case of Line 11 in Figure 2. Assume the sensitivity map knows that the branches at Line 28 and 8 are only sensitive to b_3. To force the observation of the branch at Line 28, ConcFuzz ensures the value of b_3 to be 2 (which is as same as the exploit input). Then, it mutates over b_2 and keeps the remaining bytes as same as the exploit input to get many samples that miss the branch at Line 11.

4.2 Location Ranking

Given the concentrated test-suite, the ranker first removes the duplicate traces from the test-suite to avoid biasing towards any single trace. Then, it computes the necessity and sufficiency scores of each on-exploit location v_i. It first computes these three values: 1) the number of test cases observing v_i and triggering the vulnerability ($\#(X_i = 1 \wedge C = 1)$), 2) the number of test cases triggering the vulnerability ($\#(C = 1)$), and 3) the number of test cases observing v_i ($\#(X_i = 1)$). Finally, it computes the necessity score $N = \frac{\#(X_i=1 \wedge C=1)}{\#(C=1)}$ and the sufficiency score $S = \frac{\#(X_i=1 \wedge C=1)}{\#(X_i=1)}$.

For concreteness, we revisit our running example shown in Figure 2 and the concentrated test suite generated shown in Figure 3. For v_{81}, the sufficiency score $P(C = 1|X_{81} = 1) = \frac{23}{37}$ while the necessity score $P(X_{81} = 1|C = 1) = 1$. Then, it normalizes both scores by `min-max scaling` and ranks locations according to L2-norm of the normalized necessity and sufficiency scores. The normalization function NM and L2-norm score are defined as follows:

$$NM(N) = \frac{N - min(N)}{max(N) - min(N)}, NM(S) = \frac{S - min(S)}{max(S) - min(S)}$$

$$\text{L2-norm score} = \sqrt{NM(N)^2 + NM(S)^2}$$

where $min(N)(min(S))$ is the minimum of the necessity (sufficiency) score across all branch locations (similarly $max(N)$ and $max(S)$ are defined). We use L2-norm as the ranking metric mainly because it treats the necessity and sufficiency score equally important. Note that L2-norm is just one of many reasonable scoring metrics that could be used. As the other metrics (e.g., Ochiai) use the same counts as L2-norm for computing the scores [29], we expect them to perform comparably on our concentrated test-suite.

The ranker reports the Top-K locations as localized candidates. If there are multiple locations with the same score, the ranker sorts them according to the proximity to the crash point. The closer the location to the crash location, the higher the rank.

5 IMPLEMENTATION

We implement VulnLoc on top of a binary instrumentation framework called DynamoRIO [2]. Our DynamoRIO client is written in C++ and subsequent statistical analysis is implemented offline in Python. VulnLoc consists of about 1.4 KLOC in total.

Dynamic Instrumentation. We build a DynamoRIO client to record the outcome of each branch instance. The client instruments opcodes (e.g., jle, jmp, je) which are related to conditional statements to record the address and value of each conditional statement executed in a file. We experimented with both instruction and branch level instrumentation. We selected the latter because the former could take a few minutes to hours to terminate, while the latter takes less than several seconds for our traces on average.

Input Mutation - Values. VulnLoc allows users to define their input mutation strategy. The default mutation strategy is performed on byte level which includes both single-byte mutation and pairwise mutation. Users can configure the maximum number of bytes to mutate simultaneously. VulnLoc also allows the users to optionally specify their strategies via a configuration file if they have any prior knowledge about the input format. Specifying the input format speeds up ConcFuzz by avoiding unnecessary mutations.

Input Mutation - Size. VulnLoc does not change the size of the original exploit input explicitly during fuzzing. However, ConcFuzz mutates all values including numeric length values and NULL-termination characters, implicitly changing lengths of inputs. For example, in the image processing library LibTIFF, our approach would change the image length attribute in the input, hence changing the image data size.

Vulnerability Oracle. VulnLoc allows users to define their own oracle for detecting whether an execution of a buggy program triggers a vulnerability. In our evaluation, we utilize the program crash or other detecting tools (e.g., Valgrind) as our oracle for memory safety. For numerical errors and null dereference, we dynamically instrument the binary with the additional checks.

Binary to Source mapping. Our entire analysis is independent of source code but we compare our Top-K vulnerability locations with the developer patch for validating the correctness of our results. We implement a wrapper that maps binary instructions to the corresponding source code statements for the convenience. The wrapper is built on top of `objdump` utility in Linux [3].

Table 1: Vulnerable applications for evaluating VulnLoc.

App.	Description	LOC
LibTIFF	A library for reading and manipulating TIFF files.	66K
Binutils	A collection of tools capable of creating the managing binary programs.	2.7M
Libxml2	A library for parsing XML documents.	0.2M
Libjpeg	A library for handling JPEG image format.	42K
Coreutils	A collection of basic tools used on UNIX-like systems.	63K
JasPer	A collection of tools for coding and manipulating images.	28K
FFmpeg	A collection of libraries and programs for handling video, audio and other files.	0.9M
ZZIPlib	A library for extracting data from files archived in a single zip file.	8K
Potrace	A tool for tracing bitmap images.	9K
Libming	A library for manipulating Macromedian Flash files.	66K
Libarchive	A library which manipulates streaming archives in a variety of formats.	0.1M

Optimization: Parallelization. VulnLoc uses parallelization to speed up certain tasks. In the fuzzing phase, the relationship inference is strictly sequential, however, the input mutation and execution are independent. Thus, instead of updating the sensitivity map for each test case, we dedicate each core to the fuzzing procedure of each mutation target and collect the test cases. Then, we utilize the collected test cases to update the sensitivity map once for each round of fuzzing. In the ranking phase, the sufficiency and necessity scores for multiple locations can be computed simultaneously before the ordering of L2-norm score.

Optimization: Caching. VulnLoc stores the generated inputs and their corresponding traces for each round of fuzzing. In the fuzzing phase, if ConcFuzz checks all the values of an input byte, it will avoid the mutation over the specific input byte. This significantly increases the efficiency of ConcFuzz as each execution of the vulnerable program requires a certain amount of time.

6 EVALUATION

We aim to answer the following research questions:

- **[RQ1]** How effective is VulnLoc on real-world CVEs?
- **[RQ2]** Does ConcFuzz help to prevent test-suite bias and hence over-fitting?

We select a set of real-world CVEs and run VulnLoc to generate possible vulnerability locations. We validate the efficacy of VulnLoc by comparing our results to developer patches for the CVEs as the ground truth. We extract the developer patches from the bug reports or the commits provided by the developers.

6.1 Subjects and Setup

Our subjects are chosen to satisfy three requirements:

(1) The vulnerable applications can be executed with our instrumentation platform;
(2) A working exploit is available; and,
(3) A valid developer patch is available.

Diversity in Benchmarks. We select 43 CVEs that correspond to 11 applications, shown in Table 1. Our dataset includes all 15 CVEs studied in a recent work called SENX [16] that satisfy the above three criteria[1]. We added 28 more CVEs to increase the diversity of the benchmarks, as SENX benchmarks have only 2 kinds of security vulnerabilities. Our final benchmarks have 6 categories of vulnerabilities including 26 buffer overflows (BO), 4 divide-by-zero

Table 2: Efficacy of VulnLoc for vulnerability localization in details [RQ1]. Column "#B" shows the number of branch conditions in total. "#UB" indicates the unique on-exploit locations. "Is developer patch right before crash loc?" shows whether the developer patch is right before the crash location or not. "Size(TS)" means number of unique traces generated by VulnLoc for each CVE in 4 hours. "In Top-5?" describes whether there is a correct location hitting one of the Top-5 candidates outputted by VulnLoc. "SM" means that the location of the developer patch hits one of the Top-5 candidates. "EQ" indicates the existence of an equivalent vulnerability location in Top-5 candidates. The last column describes the rank of VulnLoc's output where the developer patch or a semantically equivalent patch appears.

App.	CVE ID	Bug Type	#B	#UB	Is developer patch right before crash loc?	Size(TS)	In Top-5?	Rank
LibTIFF	CVE-2016-3186	BO	0.3K	30	✓	14	✓(SM)	2
	CVE-2016-5314	BO	0.1M	0.7K	✗	0.4K	✓(SM)	5
	CVE-2016-5321	BO	6.6K	0.6K	✓	4.3K	✗	18
	CVE-2016-9273	BO	8.2K	0.5K	✗	1.7K	✓(EQ)	1
	CVE-2016-9532	BO	21.1K	0.7K	✗	0.4K	✓(EQ)	1
	CVE-2016-10092	BO	15.4K	0.9K	✗	5.4K	✓(EQ)	3
	CVE-2016-10094	BO	41.1K	1.0K	✓	4.0K	✓(SM)	1
	CVE-2016-10272	BO	1.2M	0.9K	✗	19	✗	39
	CVE-2017-5225	BO	12.8M	0.7K	✗	3.1K	✓(EQ)	1
	CVE-2017-7595	DZ	13.1K	0.8K	✗	2.7K	✓(EQ)	1
	CVE-2017-7599	DT	10.2K	0.8K	✓	4.5K	✓(EQ)	1
	CVE-2017-7600	DT	10.3K	0.7K	✓	30	✓(SM)	1
	CVE-2017-7601	IO	13.5K	0.9K	✓	2.4K	✓(SM)	4
	Bugzilla-2611	DZ	0.1M	0.6K	✗	1.4K	✓(SM)	1
	Bugzilla-2633	BO	6.1K	0.7K	✗	5.8K	✓(EQ)	1
Binutils	CVE-2017-6965	BO	2.3K	0.5K	✓	0.4K	✓(SM)	4
	CVE-2017-14745	IO	9.5K	0.6K	✓	1.5K	✓(SM)	1
	CVE-2017-15020	BO	16.0K	1.1K	✓	1.4K	✓(SM)	1
	CVE-2017-15025	DZ	28.1K	1.0K	✓	1.4K	✓(SM)	1
Libxml2	CVE-2012-5134	BO	7.7K	1.5K	✓	20.9K	✓(SM)	1
	CVE-2016-1838	BO	0.4M	1.0K	✓	4.7K	✓(SM)	1
	CVE-2016-1839	BO	1.5M	1.4K	✗	0.9K	✓(SM)	1
	CVE-2017-5969	ND	22.5K	1.4K	✓	10.0K	✗	24
Libjpeg	CVE-2012-2806	BO	1.5K	0.2K	✓	46	✓(SM)	1
	CVE-2017-15232	ND	0.1M	0.6K	✓	8.0K	✓(SM)	1
	CVE-2018-14498	BO	1.2K	0.1K	✓	0.1K	✓(SM)	1
	CVE-2018-19664	BO	21.3M	0.1K	✗	5	✓(EQ)	3
Coreutils	GNUbug-19784	BO	0.2K	34	✓	0.5K	✓(SM)	1
	GNUbug-25003	IO	0.1K	0.1K	✓	7	✓(SM)	1
	GNUbug-25023	BO	1.3K	0.3K	✗	0.1K	✗	> 200
	GNUbug-26545	IO	0.5K	0.2K	✗	1.9K	✓(SM)	2
JasPer	CVE-2016-8691	DZ	38.9K	0.3K	✗	0.1K	✓(EQ)	1
	CVE-2016-9557	IO	44.0K	0.5K	✓	2.7K	✓(SM)	4
FFmpeg	CVE-2017-9992	BO	11.7K	0.6K	✓	0.6K	✓(EQ)	1
	Bugchrom-1404	IO	7.6M	0.9K	✗	0.4K	✗	187
ZZIPlib	CVE-2017-5974	BO	0.1K	0.1K	✗	0.2K	✓(SM)	2
	CVE-2017-5975	BO	0.1K	0.1K	✗	0.2K	✓(EQ)	2
	CVE-2017-5976	BO	0.1K	0.1K	✓	0.3K	✓(SM)	1
Potrace	CVE-2013-7437	BO	0.3M	0.1K	✗	2	✓(EQ)	1
Libming	CVE-2016-9264	BO	38	26	✗	29	✓(EQ)	4
	CVE-2018-8806	UF	1.1K	0.1K	✓	2.3K	✓(EQ)	2
	CVE-2018-8964	UF	1.1K	0.1K	✓	4.6K	✓(EQ)	5
Libarchive	CVE-2016-5844	IO	6.1K	0.7K	✓	46	✓(SM)	1

[1] SENX has 42 benchmark programs. We eliminated the following: 18 programs that do not have any developer patches (missing ground truth to evaluate against); 2 that do not have reproducible exploits, 2 that are on x86 CPUs while our present implementation supports only x64; 5 that do not work on vanilla DynamoRio without instrumentation (either crashing DynamoRio or taking hours and utilizing excessive memory for a single trace).

(DZ), 7 integer overflows (IO), 2 null pointer dereferences (ND), 2 heap use-after-free (UF) and 2 data-type overflows (DT).

The final benchmark programs have sizes ranging from 10 thousand to 2 million LOC. Most of them have very few (less than 30) or no manually written tests for the vulnerable program and its configuration. We obtained one exploit for each vulnerability from public CVE repositories or GitHub issue lists. The exploit input sizes vary from 1B to 75KB with an average of 8KB. Table 2 shows that the exploit traces have a few tens to millions of observed on-exploit branch instances. Recall that VulnLoc works by recording only branch conditionals, i.e., one per basic block. On average, there are 1 million on-exploit branch instances, with a minimum of 38 and a maximum of 7.6 million. Due to loops and recursion, many observed locations repeat—we also report the unique number of locations considered by VulnLoc for computing scores in Table 2.

Experiment Setup. All our experiments are performed on a 56-core 2.0GHz 64GB RAM Intel Xeon machine. Each round of fuzzing phase allows to mutate upto 2 bytes at a time (β=2 in Algorithm 2) and mutates γ times over each mutation target. We set γ=200 (see Algorithm 2) for the default mutation strategy. We set a timeout of 4 hours per benchmark to generate a test-suite and allow the fuzzing phase to fork 10 processes maximally.

Correctness criteria. To evaluate the correctness of identified patch locations, we use developer-provided fixes or patches in public source-code repositories of these applications as ground truth. These patches are small—most of them have between $1-3$ locations modified. We say that VulnLoc is able to pinpoint a correct or actionable location, if:

- The Top-5 locations outputted by VulnLoc coincides with (at least one) location of the developer patch; or
- A patch semantically equivalent to the developer patch can be applied at one of the Top-5 locations from VulnLoc.

We create semantically equivalent patches, where needed, using the available live variables at the highlighted location. If the predicted location is in the same function as the developer patch and all the variables used in the developer patch are live at that point, a simple displacement of developer patch usually suffices. Otherwise, we use domain-specific knowledge to create the equivalent patch using other live variables which are tightly dependent on the variables patched in the developer patch. In order to validate our patches, we run the patched application on the entire concentrated test-suite, which consists of thousands of tests, as well as the developer test suite if available. An example of an equivalent patch is described in Section 7. Our criterion of choosing Top-5 recommendations follows from empirical studies on expectations from automated fault localization tools reported by practitioners [22]. We also report the best rank of a correct location highlighted by VulnLoc in Table 2.

6.2 [RQ1] Efficacy for Vulnerability Localization

Main Results. Figure 6 summarizes the efficacy of VulnLoc for vulnerability localization and the distribution of the type of the outputted vulnerability locations. Out of 43 CVEs, VulnLoc successfully pinpoints the correct locations for 38 CVEs within the Top-5 candidates, Among these 38 CVEs, the vulnerability location for 25 CVEs hits the topmost candidate (see Figure 6). Recall that

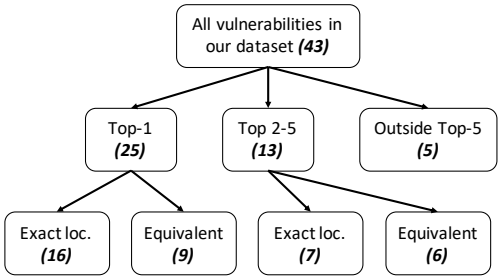

Figure 6: Efficacy of VulnLoc for vulnerability localization [RQ1].

there may exist multiple locations which are equivalent for fixing the vulnerability. We observe that, for 23 out of 38 successful CVEs, one of the top-5 candidate locations corresponds exactly to one of the locations of the developer patch. For 15 out of 38 CVEs, we can create an equivalent patch. To further investigate these results, Table 2 presents the detailed result of vulnerability localization for each CVE. VulnLoc generates a rich test-suite for each vulnerability. Unlike the manually written test-suite where no test triggers the bug, VulnLoc generates $2K$ test cases on average, around 40% of which trigger the vulnerability for half of the benchmarks. In addition, VulnLoc performs well on all categories of security bugs: It successfully identifies the correct locations in Top-5 candidates for 23 buffer overflows, 6 integer overflows, all 4 divide-by-zero, 1 null dereference, 2 heap use-after-free and 2 data-type overflows. In addition, VulnLoc performs equally on different applications. For example, it successfully identifies the correct locations for 13 out of 15 CVEs in `LibTiff`, all 4 CVEs in `Binutils`, and 3 out of 4 CVEs in `Libxml2`. This indicates that the success of VulnLoc is not correlated with the size and the type of applications.

Ruling Out Spurious Correlation. Given the statistical nature of VulnLoc, one may ask whether the results observed are an artifact of pure chance or spurious correlations, as the correlation does *not* imply causation. We additionally investigated why VulnLoc works in the cases where it reports the right candidate in the Top-5. First, we observed that the odds of pinpointing the correct branch location in the Top-5 by random chance is extremely low, as each benchmark executes thousands of basic blocks in one exploit. VulnLoc is doing significantly better than randomly guessing locations. Second, we manually investigated why VulnLoc assigns the highest score to the correct location whenever it does. To carry out this investigation, we extended VulnLoc to compute the sensitivity map for the variables around that location. We found that certain variables have the highest L2 scores—they are most sensitive to the transformation of a benign input into an exploit. We find these highly sensitive variables often correspond to the variables that are sanitized or bounded in the developer patch. For example, the variable *count* has been correctly identified as the most sensitive variable for `CVE-2016-3186`. Our manual investigation confirms that a simple extension to VulnLoc can identify a handful of candidate variables for fixing the vulnerability, beyond just identifying the correct location. This shows that the results outputted by VulnLoc is explainable and not an artifact of spurious correlations.

Performance. The total time taken for vulnerability localization on each CVE has two components: Fuzzing time and analysis time. We set the fuzzing time to 4 hours for all the CVEs. The analysis time varies with each CVE and the number of candidates to report (e.g., Top-100). The maximum analysis time taken by VulnLoc is within 10 minutes with Top-200 candidates to report.

Distance to Crash Locations. One way of vulnerability localization is to consider the location right before the crash point as the vulnerability location. In 19 out of 43 of the CVEs we study, the developer patches do not coincide with the crash location, as detailed in Table 2. Our correct locations identified by VulnLoc are on a location different from the crash location for 10 out of 38 CVEs. For these 10 CVEs, the authors could not identify a feasible patch equivalent to the developer patch at the crash location manually. An example where the localized vulnerability is far from the crash location is CVE-2016-5314, which is presented in Section 2.

Need for Probabilistic Approaches. In many CVEs (33 out of 43) the vulnerability locations do not have both necessity and sufficiency scores equal to 1, even for developer patches. Such vulnerability locations do *not* cleanly separate all exploits from benign ones. The lack of any program points, at which a clean separation between passing and failing test is possible, highlights the inherent uncertainty in choosing between candidate locations. This motivates the need for probabilistic approaches such as ours.

6.3 [RQ2] Over-fitting Reduced by Concentrated Test-suites

We examine the impact of the test-suite bias on vulnerability localization. Poor test-suites make it difficult to distinguish between different program points as vulnerability locations. This is exhibited by many locations obtaining the same score from localization. On the other hand, a concentrated test-suite segregates locations better, giving different scores to different locations. We can therefore measure how much a test-suite contributes towards segregating vulnerability locations by analyzing the final scores of the locations.

To show that a concentrated test-suite segregates locations effectively, we evaluate the effect of the following 3 kinds of test-suites, keeping our statistical analysis (Section 4.2) unchanged:

- T1: a biased test-suite which only contains exploits,
- T2: a biased test-suite which only contains tests reaching the crash location, and
- T3: a concentrated test-suite produced by VulnLoc.

We measure the efficacy of VulnLoc under these three test-suites by counting the number of branch locations which have the same score. We call a set of locations with the same score as a cluster. Note that if the test-suite is effective in vulnerability localization, the number of clusters will be very large. To measure the distinguishability of a given test-suite, we set T3 as the baseline and compute the ratio of the number of clusters generated by using T1 or T2 to the number of clusters using T3, which is called `distinguishability ratio`.

Figure 7 summarizes the distinguishability ratio of the biased test-suites T1 and T2 on vulnerability localization for 43 CVEs. For 36 out of 43 CVEs (84%), the number of clusters generated by T1 is 50% fewer than the one generated by the concentrated test-suite T3. Similar results (50% fewer clusters for 36 CVEs) are also shown

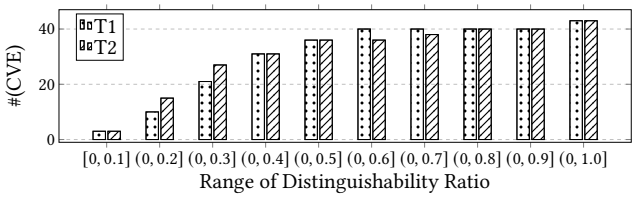

Figure 7: Effects of test-suite bias [RQ2].

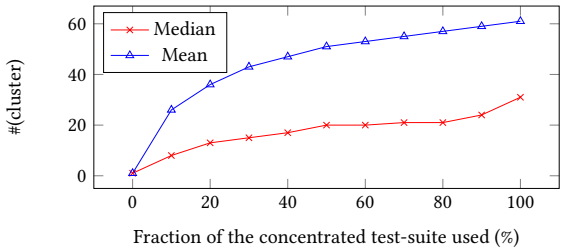

Figure 8: Effects of the size of a concentrated test-suite: As the fraction of tests utilized increases, the candidate locations become better separated (i.e. more clusters appear).

for T2. This clearly demonstrates that a concentrated test-suite (T3) improves significantly over other test-suites.

As another way of measuring the quality of a concentrated test-suite, we show how locations get better separated as we use more and more of the test-suite. Consider the number of clusters obtained after using some tests—the higher the number of clusters, the greater we can distinguish between locations. Figure 8 summarizes the average and median number of clusters generated by taking a fraction of the concentrated test-suite for 43 CVEs. For most of the CVEs, the number of clusters increases significantly with increase in the fraction of the concentrated test-suite. This directly demonstrates that utilizing more of the concentrated test-suite results in increasing segregation of locations by score.

Existing directed test-case generation tools like AFLGo [8] and F3 [18] can be used for our purpose. However, they are *not* designed to produce concentrated test-suites, which is the key conceptual advance in our proposed technique (see Section 2.3). We experimentally show both F3 and AFLGo generate test-suites which are biased towards the crash location, thus, their test-suites belong to the category T_2. Furthermore, these tools rely on external source-based analysis engines such as dynamic symbolic analysis (for F3) and intra-procedural control flow graph construction (for AFLGo).

Comparison with AFLGo. We compared our work quantitatively with the directed fuzzer AFLGo [8]. We collect all the inputs generated by AFLGo as our test-suite, with the crash location as the target. Executing the vulnerable problem with these inputs may reach or deviate from the crash location, because we consider all the tests generated by AFLGo in the process of reaching the target. Although this test-suite is balanced to some extent (and hence helps AFLGo), AFLGo can only successfully locate the vulnerability location in Top-5 for 18 out of 43 CVEs; this is also somewhat because of the complexity of partial control flow graph construction in AFLGo.

```
1   int readextension(void){
2      ...
3      char buf[255];
4      ...
5   - while ((count = getc(infile)) && count <= 255)
6   + while ((count = getc(infile)) && count >= 0 && count <= 255)
7        if (fread(buf, 1, count, infile) != (size_t) count) {...}
8   }
```

Figure 9: VulnLoc highlights the same location where the developer patch is applied for CVE-2016-3186.

In comparison, our approach indicates the vulnerability location among Top-5 candidates in 38 out of 43 CVEs in total. These results show that while our concentrated fuzzing is a form of directed fuzzing, directed fuzzing tools cannot be straightforwardly used for our problem.

Comparison with F3. We also compare with the fault localization tool F3 [18]. We keep the same ranking algorithm used in VulnLoc and only change the test-suite for a fair comparison over the quality of the test-suite. The implementation of F3 uses an out-of-date LLVM version, 2.9. Due to insufficient support of external functions and the inline assembly functions, F3 fails to generate test-suite for 19 CVEs. We do not know how F3 would have performed in localization accuracy for these 19 CVEs if the tool implementation was able to handle them. For the remaining 43−19 = 24 CVEs, the size of the test-suite generated by F3 is around 4 times smaller than the test-suite generated by VulnLoc. In our experiments, F3 always recommends vulnerability locations at or next to the crash locations. The reason is overfitting: The test-suite obtained from F3 has a high density of tests that reach the crash point. If the vulnerability location of a given CVE (e.g., CVE-2016-5314 in Section 2) is not right before the crash location, F3 fails to pinpoint the correct location within Top-5 candidates. Among the 24 CVEs that F3 handles, it localizes correctly in Top-5 for 19 out of them. In contrast, VulnLoc successfully localizes among Top-5 for all the CVEs which F3 works correctly and 3 for more (total 22).

7 CASE STUDIES

In order to understand the quality of vulnerability localization, we present two examples: a) CVE-2016-3186 for which the developer patch coincides with one of the Top-5 candidates and b) CVE-2016-8691 for which the developer patch does not coincide with any of the Top-5 candidates but there is an *equivalent* manually generated patch at one of the Top-5 candidates.

Finding developer patch location (CVE-2016-3186) This is a buffer overflow in `LibTIFF` which causes a denial of service via a crafted GIF image. Consider Figure 9, the overflow happens in function `readextension` when it reads a GIF extension block at Line 7. When `getc` detects the end of file, it returns *EOF* which is negative number. However, the loop condition only checks if `count` ≤ 255. If `count` is negative, the loop condition is satisfied and `count` is casted to `size_t`, which leads to the buffer overflow. VulnLoc analyzes this CVE and outputs the branch condition in Line 5 as one among the Top-5 candidates. This coincides exactly with the developer patch which adds an additional check at Line 5 to prevent overflow.

```
1     ...
2     samplerate_idx = (flags & MP3_SAMPLERATE) >> MP3_SAMPLERATE_SHIFT;
3   + if (samplerate_idx < 0 || samplerate_idx > MP3_SAMPLERATE_IDX_MAX){
4   +    error("invalid samplerate index");}
5     ... // <-- code with no relation to samplerate_idx
6     samplerate = mp1_samplerate_table[samplerate_idx];
```

Figure 10: Developer patch for CVE-2016-9264.

```
1     ...
2     samplerate_idx = (flags & MP3_SAMPLERATE) >> MP3_SAMPLERATE_SHIFT;
3     ... // <-- code with no relation to samplerate_idx
4   +  if (samplerate_idx < 0 || samplerate_idx > MP3_SAMPLERATE_IDX_MAX){
5   +     error("invalid samplerate index");}
6     samplerate = mp1_samplerate_table[samplerate_idx];
```

Figure 11: Semantically equivalent developer patch at the location highlighted by VulnLoc for CVE-2016-9264.

Finding equivalent patch location (CVE-2016-9264) This is an example of an out-of-bounds read in `Libming` library which can crash any web application that uses this library to process untrusted mp3 files. Consider Figure 10, the variable `samplerate_idx` in Line 2, is read from an input mp3 file and is used to set the `samplerate` in Line 6. Executing the exploit mp3 file results in an out-of-bounds access at Line 6 which sets `samplerate` to 0 and later results in a crash due to floating-point exception. So, the developer patch is applied at Line 3 just after reading `samplerate_idx` from input. However, VulnLoc suggests to add a check just before the out-of-bounds access at Line 6, shown in Figure 11. The original code between Line 2 and Line 6 does not use `samplerate_idx` and it is not affected by the input file.

8 RELATED WORK

One of the earliest efforts in fault localization is via dynamic slicing [5] or data dependency analyses [11, 27]. It takes in a program input and a slicing criterion in the form of $\langle l, v\rangle$ where l is a location and v is a variable. It uses data and control dependencies to explain the value of v in l in the execution trace of the given input. Such analyses often reason about explicit dependencies only, which are dependencies that manifest in the executed trace. Further, since dynamic slicing involves high computational overheads and dynamic slices are large, more accurate methods to localize observable errors in programs have been studied. One of the notable works in this regard is the principle of delta debugging [40] which localizes observable errors by computing the differential of a failing artifact and a "similar" benign artifact. The artifact could be in the form of test inputs or execution traces. Such analysis can reason about dependencies that may not be visible in the original exploit trace. One of the major difficulties in employing this line of work is that its accuracy crucially depends on the choice of the benign artifact.

Progress in localization via trace comparison has led to other works involving a more systematic generation of the benign trace, and a natural extension to probabilistic reasoning. These include the use of a systematic off-line search to generate the passing trace via branch direction mutation [36], as well as online predicate switching by forcibly switching a branch predicate's outcome at run-time [42]. Instead of forcibly changing a branch predicate at run-time,

directed fuzzing generates inputs with the goal of flipping branch predicate(s). So, our statistical analysis reasons about the original program without modified logic, unlike predicate switching.

Our work follows the statistical fault localization framework [38], where a score is assigned to each statement of the program based on its occurrence in passing and failing execution traces. One of the first works in this regard is Tarantula [19], which has subsequently been followed by many works proposing many scoring metrics, including the Ochiai metric [4]. The main hypothesis in these works is that the control flow of the execution traces of tests can be used to determine likely causes of failure of a test. Thus, if a statement occurs frequently in failing test executions and rather infrequently in passing test executions, it is likely to be scored highly and brought to the attention of the developer. It is well-known that the accuracy of these methods is highly sensitive to the choice of tests [28, 35]. Most works in this regime use externally provided or arbitrarily chosen test suites.

Very few works have attempted to address the central challenge of choosing the right test suite. F3 is the most closely related work to ours as it targets synthesizing test-suites for statistical localization [18]. It builds on the techniques proposed in BugRedux [17]. The goal of BugRedux is different from ours, it is to re-produce a field failure trace by following through "breadcrumbs" given as locations visited. F3 [18] relaxes the execution synthesis component of BugRedux by generating many tests via symbolic execution. We experimentally compared concentrated fuzzing with F3 in this paper, demonstrating that concentrated fuzzing has better performance. Another difference is that our work makes fewer assumptions and avoids complex symbolic analysis. Symbolic analyses have well-known challenges in scaling to large programs, especially on binaries [33]. Experimentally, we find that F3 failed to handle 19/43 benchmarks we tested. We are aware of an independent and concurrent work called AURORA, which also proposes statistical localization under similar assumptions [7]. However, it reuses an off-the-shelf fuzzing strategy, namely AFL's crash exploration mode. AURORA proposes additional mechanisms for synthesizing and ranking particular kinds of logical predicates during its statistical analysis. In contrast, VulnLoc only devises a new systematic test-suite generation technique, while retaining the rest of the structure of statistical fault localization entirely. Our techniques are thus complementary as one could combine our concentrated test-suite generation with the predicate synthesis and ranking mechanism proposed in AURORA.

A different line of work employs symbolic analysis methods for localizing the root cause of an observable error [9, 12, 20, 30]. The central observation in these works is that localization can benefit from specification inference. Even in the absence of formal specifications of intended program behavior, these works seek to infer properties of intended program behavior by symbolically analyzing various program artifacts such as failing execution traces, past program versions as so on. These approaches proceed via source code analysis, and incur the overheads of symbolic execution.

Our specific proposal for concentrated fuzzing is perhaps most closely related to GREYONE, a faster taint-based fuzzing for bug-finding [13] which extends notions of taint or influence from recent work [10]. Concentrated fuzzing has completely different objectives to this work, as it does not aim to maximize coverage or exploits.

SENX is an automatic patch synthesis tool for certain vulnerabilities based on information from source code and an exploit [16]. SENX uses a simplistic strategy for localization: It uses the program point where the safety property is violated as the vulnerability point, most of which are right before the crash location. Such localization is typically only sufficient for if-guard fixes at the crash location, which may not fix the fault in a general way, but workaround to prevent an error from being observable. In our experiments, we have reported 10 (out of 38) correct vulnerability locations which are different from the crash location. We show an example in Section 2.

Other works that aim to localize by identifying workarounds that make errors unobservable have also been proposed, such as Talos [15]. Talos extensively uses source code and specializes for specific software coding practices or idioms. A number of prior works use source code for vulnerability localization, including a recent work that employs deep learning over code features [24]. Our work minimizes assumptions about the availability of such features and yet achieves high accuracy in real-world programs.

9 CONCLUSION

In this paper, we propose a novel approach of combining directed test-generation techniques with statistical localization. Our approach takes a principled view of the problem, namely, synthesizing input distributions that have a balanced proportion of samples observing (as well as avoiding) candidate locations. Our work is the first to propose the use of directed fuzzing for statistical localization. Our specific procedure for directed fuzzing, called ConcFuzz, can be of independent interest in different applications. Our tool VulnLoc works directly on binaries given an exploit. VulnLoc exhibits promising localization accuracy in large real-world applications for several types of vulnerabilities. More information about VulnLoc can be found on the project page at https://github.com/VulnLoc/VulnLoc.

10 ACKNOWLEDGMENTS

We thank Shruti Tople, Shweta Shinde, Shin Hwei Tan, Teodora Baluta, Ahmad Soltani and the anonymous reviewers for helpful feedback on this work. We thank Jinsheng Ba for helping us in experiments. All opinions expressed in this paper are solely those of the authors. This work is supported by the Crystal Center at National University Of Singapore and the research grant DSOCL17019 from DSO, Singapore. It was also partially supported by the National Satellite of Excellence in Trustworthy Software Systems and funded by National Research Foundation (NRF) Singapore under National Cybersecurity R&D (NCR) programme.

REFERENCES

[1] [n.d.]. Honggfuzz: Security oriented software fuzzer. https://honggfuzz.dev/.
[2] 2019. DynamoRIO: Dynamic Instrumentation Tool Platform. https://www.dynamorio.org.
[3] 2019. objdump. https://linux.die.net/man/1/objdump.
[4] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION.*
[5] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. In *PLDI.*
[6] Fatmah Assiri and James Bieman. 2016. Fault localization for automated program repair: effectiveness, performance, repair correctness. *Software Quality Journal* (2016).

[7] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *USENIX*.

[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *CCS*.

[9] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic Debugging. In *ICSE*.

[10] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. 2019. One Engine To Serve'em All: Inferring Taint Rules Without Architectural Semantics.. In *NDSS*.

[11] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. {REPT}: Reverse Debugging of Failures in Deployed Software. In *OSDI*.

[12] Evren Ermis, Martin Schäf, and Thomas Wies. 2012. Error invariants. In *International Symposium on Formal Methods*.

[13] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. (2020).

[14] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. (2021).

[15] Zhen Huang, Mariana DAngelo, Dhaval Miyani, and David Lie. 2016. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *S&P*.

[16] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using Safety Properties to Generate Vulnerability Patches. In *S&P*.

[17] Wei Jin and Alessandro Orso. 2012. BugRedux: reproducing field failures for in-house debugging. In *ICSE*.

[18] Wei Jin and Alessandro Orso. 2013. F3: fault localization for field failures. In *ISSTA*.

[19] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *ICSE*.

[20] Manu Jose and Rupak Majumdar. 2011. Cause clue clauses: error localization using maximum satisfiability. In *PLDI*.

[21] Samuel T King and Peter M Chen. 2003. Backtracking intrusions. In *SOSP*.

[22] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *ISSTA*.

[23] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (2019).

[24] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In *ISSTA*.

[25] Zhenkai Liang and R. Sekar. 2005. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *CCS*.

[26] LibTIFF. [n.d.]. an open source implementaiton on github. https://github.com/vadz/libtiff.

[27] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software.. In *NDSS*.

[28] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *ICSE*.

[29] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *ICSE*.

[30] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. 2012. Darwin: An approach to debugging evolving programs. *TOSEM* (2012).

[31] Jeremias Röβler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. 2012. Isolating failure causes through test case generation. In *ISSTA*.

[32] Cliff Saran. 2018. Security professionals admit pais getting harder. https://www.computerweekly.com/news/252438578/Security-professionals-admit-patching-is-getting-harder.

[33] E. J. Schwartz, T. Avgerinos, and D. Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *S&P*.

[34] Kelly Sheridan. 2018. It Takes an Average 38 Days to Patch a Vulnerability. https://www.darkreading.com/cloud/it-takes-an-average-38-days-to-patch-a-vulnerability/d/d-id/1332638.

[35] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex–white box test generation for. net. In *TAP*.

[36] Tao Wang and Abhik Roychoudhury. 2005. Automated path generation for software fault localization. In *ASE*.

[37] Tielei Wang, Chengyu Song, and Wenke Lee. 2014. Diagnosis and emergency patch generation for integer overflow exploits. In *DIMVA*.

[38] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *TSE* (2016).

[39] Michal Zalewski. [n.d.]. american fuzzy lop. https://github.com/google/honggfuzz.

[40] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *TSE* (2002).

[41] Mingwei Zhang, Aravind Prakash, Xiaolei Li, Zhenkai Liang, and Heng Yin. 2012. Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis. In *NDSS*.

[42] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Pruning dynamic slices with confidence. In *PLDI*.